The Dissertation Committee for Vincent Liu
certifies that this is the approved version of the following honors thesis:

A Compiler Alternative to Expression Templates

Committee:

Calvin Lin, Supervisor

Kathryn McKinley

Don Batory

A Compiler Alternative to Expression Templates

by

Vincent Liu

HONORS THESIS

Presented to the Faculty of the Computer Science Department of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

HONORS DEGREE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2010

A Compiler Alternative to Expression Templates

Vincent Liu, B.S.
The University of Texas at Austin, 2010

Supervisor: Calvin Lin

Container classes in object-oriented languages often suffer from the problem of compiler-generated temporaries. Storage is allocated for each intermediate result in each expression, introducing huge penalties to performance. Loop fusion and array contraction, the most closely related compiler optimizations to this problem, are not powerful enough to optimize the container classes. The current workaround is to perform a manual optimization called Expression Templates (ETs), which utilizes C++ templates to perform arbitrary code generation, but is a very messy and inefficient way of doing things. We present a compiler alternative to ETs that uses programmer annotations to eliminate temporaries through expression fusion. Our solution is much better for programmer productivity than ETs, and also achieves results that are slightly better.

Table of Contents

Chapter 1

Introduction

Container classes are an abstraction for dealing with a variety of applications, from matrix libraries in scientific computing to graph representations for modeling. For many of these uses, datasets are large, and peak performance is essential. In fact, it is not uncommon for library writers to manually specialize their code for specific machines and compilers, or to go so far as to hand-code assembly for frequently-used functions. In this way, programmers sacrifice readability and good design principles for good performance. Manual optimizations are time-consuming, do not scale, and do not always compose well with compiler optimizations. There will always be messy, manual optimizations beyond those that compilers routinely perform, but commonly-used optimizations should be automated in order to amortize their cost over each use of the compiler.

For example, one of the largest sources of performance loss in matrix programs written in C++ (e.g., Blitz++ [16] and HTAs [4]) is the creation of compiler-generated temporaries for intermediate values in an expression. Each arithmetic operator will generate its own temporary and loop nest, creating unnecessary overhead and destroying cache locality in expressions involving

multiple operators. This problem is one reason why matrix library writers are reluctant to move toward higher-level languages like C++. It has a major impact on performance, and every C++ matrix library must deal with it. This study concentrates on matrix libraries and the fusion of expressions operating on those libraries.

The most common way to remove this source of performance loss is through a manual optimization technique called Expression Templates (ETs) [15], which makes use of the fact that C++ templates can generate arbitrary code at template instantiation-time. As programmers move toward higher-level languages, they trade low-level control for useful abstractions. ETs are effort to regain some of this control, but it is counterproductive. They use a syntax that was never intended to express optimizations and which, in some cases, makes the resulting code is less readable than it would be in a lower level language. Consequently involve fundamental changes to the class and cause modifications to the operand to be very difficult. The template mechanism itself was not created with arbitrary calculations in mind, and so it is inefficient at compile time and highly dependent on the compiler implementation.

A major reason why these types of manual optimizations are so common within domains such as scientific computing and modeling is that many optimizations that are well-known within each domain are not applicable for general programs. The compiler-generated temporaries can be removed through loop fusion, but traditional loop fusion is not powerful enough to apply to container classes, which are encapsulated within potentially complex operator

functions. Operators can contain arbitrary code external to the loops themselves or use opaque iterators, which require extra information to determine the validity of fusion and contraction.

We present a compiler alternative to manual solutions like ETs that utilizes domain-specific information and programmer annotations to perform expression fusion. This extra information gives the compiler knowledge about the use of member variables and the structure of operations that can enable a wealth of compiler optimizations that were previously impossible. Our compiler uses this information in conjunction with dataflow analysis to build expression trees and fuse them into a single function. With this system, programmers can program in a clean, easy-to-read, and extensible way, but still get less overheads, a reduced number of temporaries, and therefore better cache utilization.

We have implemented this compiler optimization, and tests have shown performance similar to that of Expression Templates with more than double the performance of C++ without fusion and contraction. With minimal extensions, this compiler optimization can also be applied to other types of container classes including sparse matrices and sets. The same tools can also be used to implement any number of domain specific optimizations that utilize mathematical properties of the various container classes.

Chapter 2

Background

For this study, we will be looking at container classes in C++. We assume that these are object-oriented structs or classes that encapsulate an allocation of data, a few member variables, and functions that perform operations on the data. Operations are performed by iterating over the encapsulated data, and these functions are the only allowed modifier of the data. There are other models for working with these types of classes [3], but our decomposition of these classes is fairly general and applicable to many of these other models as well. In the remainder of this thesis, container classes refer to the classes themselves or their definitions, container objects refer to individual instances of a container class, and operators refer to the functions that operate on contained data. Loop fusion will be used in reference to existing methods of fusion that operate on adjacent matrices, while expression fusion refers to the more complicated fusion of operators over container classes.

2.1 Metaprogramming

Metaprogramming refers to the process of using programs to create or modify other programs. This very general concept can be done in a vari-

ety of different ways and with a variety of different applications. Well-known concepts such as reflection and string evaluation are examples of metaprogramming tools that exist in a large number of existing languages. These tools can sometimes facilitate programmers with functionality that is difficult to express otherwise, but at a cost to the effectiveness of the compiler (especially when the generation or modifications occur at runtime) and to the readability of the code.

A very important application of metaprogramming is its ability to essentially add language features that are not inherent to the language [5]. A sufficiently powerful metaprogramming tool can implement anything from persistent storage to generics to lazy evaluation.

2.2 Template Metaprogramming

C++ contains an unexpectedly powerful metaprogramming mechanism in the form of templates. Templates allow programmers to parametrize types based on other types or on a value. Every unique specialization of the template is then instantiated at compile time by essentially creating a new type. Templates were originally introduced with the intention of providing for generics, but the way in which they were designed allows for much more than just generics - they are Turing complete [17].

A commonly cited example of the power of templates is the compile-time calculation of a factorial:

```
template <int n>
```

```
struct factorial
{
    enum{ result = n*factorial<n-1>::result };
};

// base case
template <>
struct factorial<0>
{
    enum{ result = 1 };
};

void foo()
{
    std::cout << factorial<4>::result;
}
```

This method of computing the factorial function does all of the calculation at compile time, and the generated assembly code will print out the literal integer 24. From this example, we can see a glimpse of the potential of templates. The nature of the tool provides something similar to recursion, and when specialized on a boolean value, templates can implement conditionals as well, providing the necessary requirements for Turing completeness. Another thing this example displays is one method of optimizing programs using templates. The result of compilation contains no arithmetic operations, thereby significantly reducing the run time of the program. However, this optimization comes at a cost as the calculation is simply moved to compile time where it is executed without any debugging support and without any optimization. Additionally, the code hides a subtle point that makes the type system no longer decidable. Trying to instantiate the factorial template with a negative inte-

ger will theoretically result in an infinite loop within the compiler. In reality, compilers put an upper bound (the standard recommends 17) on the depth of templates. This means that, while the type system remains decidable, it may reject 18 factorial and above, depending on the compiler implementation.

Despite these and other limitations of templates, some programmers continue to insist on using template metaprogramming because there are certain features that are not easily expressible using other methods. Within C++ matrix libraries, techniques to get rid of compiler generated temporaries are so important to performance that template metaprogrammers have developed a technique that generalizes code generation for matrix expressions called Expression Templates.

2.3 Expression Templates

The Expression Templates technique is a code generation technique for C++ matrix libraries that was proposed by Todd Veldhuizen [15] and accomplishes optimizations by specializing expressions at compile time for each different composition of operands. The Expression Template technique advocates the use of a specific class structure where both expressions and single matrices are parameterized versions of an expression superclass. In effect, the expression is turned into an object using templates. Consequently, when an expression involving an arbitrary number of terms is written, the operator returns a new expression object. The creation of this new object may or may not require the recursive creation of new template classes, but eventually the

compiler will deduce the full type of the expression, which will contain the parse tree of the expression as the template argument.

A relatively small excerpt of the code is shown in Listing 2.1. For the purposes of this thesis, we only discuss addition, but other operators are almost identical. The listing shows the binary expression template, which when one of its elements is accessed, recursively calls *apply*() on its subexpressions. The listing also shows the add functor, which implements the base case *apply*() for additions and is declared inline so that the expression will be inlined into a single function. Lastly, the listing includes the *operator* + () functions that build the expressions. Note that there is much repetition since each possible permutation of operands must be accounted for.

Listing 2.1: An excerpt from Todd Veldhuizen's original expression template library [15]. Only code directly related to addition is shown here

```
// the binary expression template
template<class A, class B, class Op>
class DVBinExprOp {
        ...
    double operator[](int i) const
    { return Op::apply(iter1_[i], iter2_[i]); }
    ...
};
// the add function-object
class DApAdd {
public:
    DApAdd() { }

    static inline double apply(double a, double b)
    { return a+b; }
};
// the five possible expression combinations
DVExpr<DVBinExprOp<double*,double*,DApAdd> >
operator+(const DVec& a, const DVec& b)
{
    typedef DVBinExprOp<double*,double*,DApAdd> ExprT;
```

```
    return DVExpr<ExprT>(ExprT(a.begin(),b.begin()));
}

template<class A>
DVExpr<DVBinExprOp<DVExpr<A>,double*,DApAdd> >
operator+(const DVExpr<A>& a, const DVec& b)
{
    typedef DVBinExprOp<DVExpr<A>,double*,DApAdd> ExprT;
    return DVExpr<ExprT>(ExprT(a,b.begin()));
}

template<class A>
DVExpr<DVBinExprOp<double*,DVExpr<A>,DApAdd> >
operator+(const DVec& a, const DVExpr<A>& b)
{
    typedef DVBinExprOp<double*,DVExpr<A>,DApAdd> ExprT;
    return DVExpr<ExprT>(ExprT(a.begin(),b));
}

template<class A, class B>
DVExpr<DVBinExprOp<DVExpr<A>,DVExpr<B>,DApAdd> >
operator+(const DVExpr<A>& a, const DVExpr<B>& b)
{
    typedef DVBinExprOp<DVExpr<A>,DVExpr<B>,DApAdd> ExprT;
    return DVExpr<ExprT>(ExprT(a,b));
}

template<class A>
DExpr<DBinExprOp<DExprLiteral, DExpr<A>, DApAdd> >
operator+(double x, const DExpr<A>& a)
{
    typedef DBinExprOp<DExprLiteral, DExpr<A>, DApAdd> ExprT;
    return DExpr<ExprT>(ExprT(DExprLiteral(x),a));
}
```

The specifics of the above code are not important, but what the reader should note is the awkwardness of the syntax and the amount of repetition in the code (an overloaded operator for each permutation of types). For a simple vector expression such as A+B+C and the simple Expression Template vector library, the creation of the expression class causes a binary expression class

be instantiated to represent an addition of an expression and C (the order is determined by associativity). The expression in turn creates another binary expression class to represent the addition of A and B, which results in the final addition operator returning the following type:

```
DVExpr<
  DVBinExprOp<
    DVExpr<
      DVBinExprOp<DVec::iterT, DVec::iterT, DApAdd> >,
    DVec::iterT, DApAdd> >
```

The innermost DVBinExprOp template class represents the addition of A and B, which are each represented as iterators. This addition is encapsulated in a generic expression (DVExpr) object, which is then passed to the other addition in this expression. Note that this expression is not immediately evaluated when it is created. In order to evaluate the expression object, a programmer would call an evaluate function on individual elements of the result, which in an ET class is an inline function where most of the code generation takes place.

With this structure, it is possible for programmers to implement any optimization that relies on the structure of the expression since the entire parse tree is available to the expression object. As previously mentioned, one can defer evaluation of the expression until it is used in an assignment or other such operation, making lazy evaluation one language feature that is innate to the ET technique. By far the most common and straightforward use of ETs is for expression fusion. By implementing the evaluation functions

in a completely straightforward manner, the lazy evaluation and per-element evaluation inherent to the ET technique easily result in fusion and contraction. For example, the following implementation of the per-element evaluation of addition shown above:

```
static inline double apply(double a, double b)
        { return a+b; }
```

The structure of the ET class will cause uses of the above A+B+C expression object's apply function to inline a+b+c into any use of the result's element through the following process:

1. `result[index]`

   is equivalent to (the type is simplified for simplicity, and the matrix types are replaced with the matrices that they represent for readability):

2. `DVBinExprOp<DVBinExprOp<B, C, DApAdd>, D, DApAdd>[index]`

   The outer array subscript operator is translated to become:

3. `AddOp.apply(BinExpr<B, C, AddOp>[index], D[index])`

   The inner array subscript operator is similarly translated to become:

4. `AddOp.apply(AddOp.apply(B[index], C[index]), D[index])`

   And finally the *apply* functions are inlined to generate:

5. `B[index] + C[index] + D[index]`

Although Expression Templates can theoretically be used for any optimization that involves optimization of expressions, the main use has been for expression fusion. Libraries have also used them to detect and use optimized functions for stencil operations and other special cases (operations on matrices of low dimensions or mathematical identities).

Expression Template libraries have been able to use expression fusion and other optimizations in order to achieve execution times that are less than half of those of the equivalent non-ET C++. Blitz++, a highly tuned ET library, has even achieved performance comparable to that of FORTRAN 90 and 77 for array computations.

Chapter 3

Problems with Expression Templates

Although Expression Template classes can achieve excellent performance, they are far from an ideal solution. They grant a lot of power to programmers, but they are detrimental to programmer productivity, they hurt the scalability of the code, they are not ideal for performance optimization, and they introduce many problems during compilation. The ideal solution, and the goal that our system strives toward, is an environment where the programmer can create a "clean" program that is intuitive and easy to understand while performance concerns are separated into a compiler optimization. Below is a more detailed discussion of the problems with Expression Templates in the context of matrix libraries, although similar arguments can apply to any container class that uses the technique. Where applicable, comparisons are against ideal solutions (hand-optimized assembly in the case of performance) and "clean" code (in the case of programmer productivity), which is assumed to implement a simple dense matrix library without ETs.

3.1 Productivity

One of the main reasons some library writers have moved to C++ instead of a lower-level language like FORTRAN is for increased programmer productivity. However, the use of Expression Templates gives rise to many of the same issues that a higher-level language is supposed to mitigate as well as a few additional issues. These include:

- ETs introduce performance concerns into the basic structure of the classes as well as the implementation.
- They unnecessarily add extra complexity to the type hierarchy.
- In order to achieve good performance, the programmer must manually ensure certain properties (the ability to inline), and these properties may require compiler expertise or be implementation dependent.
- There are restrictions on what an operation can do (e.g., operations must be independent with regards to each element and expressions cannot be modified before they are evaluated)
- Significantly increased compile-times slow down debugging and software development cycles.
- Errors in computations that are moved to compile time are extremely difficult to debug. Not only is the syntax much harder to understand, the compiler is not built to provide helpful information in the case that

the type system contains errors or becomes infinite. Normal debugging methods are useless for these types of bugs, and compiler error messages are cryptic.

The way ETs work is subtle and notoriously hard to reason about. A couple years after the concept of ETs were first introduced, problems were discovered involving incomplete alias detection, which was leading to register spillage [2]. It took 8 years for a programmers to find a solution to the aliasing problem [7], not because the solution was complicated, but because the mechanisms that make ETs work are very subtle.

3.2 Scalability

One part of programmer productivity is scalability, or the ease of adding or modifying a section of the code. The following scenarios are based on adding a feature that affects operations to a Expression Template library. Features that do not affect operations would be the same in both cases. Modifications are similar to adding features because the code growth due to addition of a feature is roughly equivalent to the amount of code affected by the feature.

Feature additions that are harder with ETs:

- Data Types - Adding another type of matrix can greatly increase code size because we need to duplicate all operations for the new type combinations, which increase as a polynomial function $O(m^n)$, where m=# of types, n=max # of operands to any operation

- Special cases - Special cases that either must be handled separately or are differentiated for performance reasons can sometimes be as bad as adding another data type (e.g., making a special function for one-dimensional matrices).

- Types of Operations - Adding, say, unary operations or scalar (as opposed to vector or matrix) operations can increase code size as well.

Feature additions that are the same in both ET and clean code:

- Operations - Added operations will have to be added in each data type the same way they are changed in the clean code

- Matrix size/dimension - If programmed correctly, an ET library can handle any size or dimension

Many of the ET scalability problems can be circumvented with additional levels of metaprogramming techniques, typically in the form of macros or programs that generate all operator implementations for each combination of types. These techniques can ease the burden on the programmer but not the underlying code bloat.

3.3 Performance

Although Expression Templates can provide greatly improved performance over non-Expression Template matrix libraries, they still have several inefficiencies compared to hand-optimized code.

- Cross equation fusion is not possible (i.e., if the programmer writes Temp=B+C; A=Temp+D, they will receive no performance benefit). This also means that programmers need to know that manual common subexpression elimination is detrimental to performance (i.e., Common=A+B; Z=Common+C; Y=Common+D is generally slower than Z=A+B+C; Y=A+B+D).

- The executable must contain duplicated template classes and functions for each of the possibilities, which can significantly increase executable size.

- ETs might not be portable. Because the performance concern is a basic part of the structure of the code, if there is a case where lazy evaluation is not beneficial, ETs cannot be removed from the library.

3.4 Compiling

Combining or interleaving compiler optimizations can be very profitable, but moving the fusion optimization to the initial template instantiation phase gets rid of this possibility. More specifically, common subexpression elimination, domain-specific simplifications, and any optimization that compares types are not possible but potentially beneficial. In addition, the class and function duplication results in significantly increased compile times since large numbers of duplicated functions must be analyzed and transformed.

Expression Templates do provide some opportunities for synergy, how-

ever, as the specialization of each function for a specific type of matrix (for instance creating a new template type for each unique matrix dimension) provides opportunities for constant propagation that did not exist before. Library writers have used knowledge of constants in these cases for optimizations like loop unrolling.

Support for Expression Templates and Template Metaprogramming is still very compiler-dependent, partly because the standards do not guarantee correct operation all of the time. The ISO C++ standard guarantees a template recursion depth of 17, and anything above this limit may fail to compile. The inline declarations are merely suggestions and do not actually guarantee inlining. The same is true for claims of metaprogrammed loop unrolling (the technique is to simply make loop bounds low and constant to encourage the compiler to do the work).

Chapter 4

Loop Fusion and Array Contraction

Loop Fusion is an optimization that is closely related to expression fusion. It takes two adjacent loops with compatible loop headers and fuses them into a single loop, which results in an increase in performance from the removal of loop overheads. The performance gain from this removal of a loop is relatively small, but the principal benefit of loop fusion is that it provides opportunities for better cache usage. If the two original loops contained references to the same object and there are no dependencies, then fusing the loops will most likely result in less cache misses.

Another potential benefit is the opportunity for array contraction, which involves the removal of array temporaries. If the first loop computes an array temporary and the second loop uses it, then it may be possible to convert the array temporary accesses into a scalar temporary. This is possible when the loops are fused in such a way that the computation and corresponding use are fused into the same iteration.

Loop fusion and array contraction can be used to eliminate compiler generated temporaries, but current implementations of these two optimizations are not powerful enough to perform expression fusion on arbitrary container

classes. Current implementations assume candidates are adjacent or at least require that intermediate code is movable, which is not always the case in modern container classes. Most implementations also do not try to handle interprocedural loop fusion even though a core property of container classes is that operations are encapsulated within separate functions.

Chapter 5

Related Work

In order to optimize container classes, particularly in the case of removing compiler-generated temporaries, many approaches attempt to provide Turing complete code generation. Template metaprogramming and Veldhuizen's Expression Templates [15] are included in this category and are by far the most common method of removing compiler-generated temporaries in C++ container classes. For the reasons listed above, this solution is far from ideal.

Similar solutions forgo template metaprogramming and instead rely on language extensions to provide powerful code generation capabilities for C++. Chiba presented a Meta Object Protocol that does compile time code generation in an object oriented manner [5]. Poletto et al. take a slightly different approach in `C and tcc by allowing for dynamic code generation and encapsulating these bits of dynamic code in special constructs [12]. Both of these approaches extend the core language to include arbitrary code generation and require rewriting or writing alternate versions of the container class. As such, these approaches still do not separate concerns.

Loop fusion is a well-known optimization that has a large amount of work associated with it. Loop fusion has been shown to provide increased

cache locality in many cases and reduce the overhead of both parallel and sequential loops [9][13][14]. Gao et al. describe an algorithm for determining which loops to fuse to maximize array contraction [6], but like most previous loop fusion work, they assume simple, adjacent loops. Even if the operator was inlined, their algorithm cannot handle operators that include code outside the loop, use some technique that makes conformability of loop bounds difficult to verify (e.g., iterators), or rely on recursion.

Lewis et al. presented a method to automatically perform fusion for contraction that is able to detect the validity of reordering the loop to enable additional contraction [11]. They present the algorithm from the view of an array language where the specific indexes that are accessed are known and do not discuss the more complex cases listed above. Their techniques are difficult to apply to very general languages like C++ and do not work for other container classes.

Chapter 6

Our Solution

We have implemented a compiler optimization using the LLVM compiler infrastructure [10] that is able to perform expression fusion and contraction on container classes like dense matrix libraries. This compiler optimization relies on programmer annotations to determine when to fuse and whether fusion is valid for a given case. Unlike many of the previously discussed optimization techniques, these annotations do not affect semantics and therefore can be easily added to existing code or removed for debugging purposes. The annotations are inline with the source code in order to make effects visually clear to the programmer and because, even though the source code is independent of the annotations, the annotations are heavily dependant on the source code.

With the complexity of current container classes and operations, expression fusion is a nontrivial optimization. ETs are only able to perform the transformation with extensive, implicit hints from the programmer. Simply the act of using ETs to implement a given class and its operators implies that the class is a container class and the operators are element-wise operations without side-effects and without any possibility of dependencies within an ex-

pression. Programmers also provide explicit information when writing an ET class including what code to generate, where the code should be generated, and where lazy evaluation is legal. There are certain pieces of information that are necessary in order to ensure the correctness of expression fusion, and the compiler optimization obtains this information through programmer annotations.

6.1 Annotations

Two key observations motivate the design of our annotation system:

1. Operators generally contain a large amount of duplicated traversal code, which includes the loop that is to be fused. The part of the operator function that actually computes the result and is unique to the operator is therefore relatively small.

2. The dimensions and other properties of the temporaries are based on the operands to the operator and can therefore be related to the operands.

The annotation system is shown in Listing 6.1 and requires two types of annotations: function annotations and essential operation markers. Annotations begin with *#pragma expfusion*, which is followed by a semicolon separated list of sub-annotations. A few functions from simple example matrix library and their associated annotations are shown below. The function annotations are placed above each constructor and operator and provide the

compiler optimization with information on what the function reads/modifies as well as the properties of the result.

Listing 6.1: An excerpt from a clean and annotated matrix library

```
struct matrix{
        public:
                matrix(int row, int col);
                matrix operator+ (const matrix& op2);
                matrix& operator= (const matrix &rhs);
                ...
        private:
                int col;
                int row;
                double *data;
};

#pragma expfusion temp_constructor; assign:return;
                property:col=col,row=row
matrix::matrix(int row, int col){
        data = (double*) malloc(sizeof(double) * row * col);
        this->row = row;
        this->col = col;
}

#pragma expfusion assign:this,return; access:rhs;
                property:col=rhs.col,row=rhs.row; shape:rhs
matrix& matrix::operator= (const matrix &rhs){
        for(int i = 0; i < row; i++){
                for(int j = 0; j < col; j++){
                        #pragma expfusion essentialop_begin 1
                        data[i*col + j] = rhs.data[i*col + j];
                        #pragma expfusion essentialop_end 1
                }
        }
}

#pragma expfusion assign:return; access:this,op2;
                property:col=this.col,row=this.row; shape:this,op2
matrix matrix::operator+ (const matrix& op2){
        matrix result = matrix(row, col);

        for(int i = 0; i < result.row; i++){
                for(int j = 0; j < result.col; j++){
                        #pragma expfusion essentialop_begin 1
```

```
                        result.data[i*result.col+j] =
                                this->data[i*result.col+j]
                                + op2.data[i*result.col+j];
                        #pragma expfusion essentialop_end 1
                }
        }
        return result;
}
```

We treat constructors much like operators because they write to a container class whose properties can be inferred from the parameters. The case above acts like an operator that does not access any other matrices, but other types of constructors also have operator analogues (e.g., copy constructors will have similar annotations to the equals operator above). The only fundamental difference is a possible *temp_constructor* annotation, which denotes the constructor that should be used for creating compiler-generated temporaries. A limitation of the current system is that one and only one constructor is defined as the *temp_constructor*, which may not be the case if different operators need different temporaries. All other function annotations apply to both constructors and operators.

The *assign* annotation denotes the container classes to which the function writes the result while the *access* annotation denotes the container classes that the function reads from. Both annotations consist of their respective keyword followed by a comma-separated list of container-class objects, which can include parameters, *return* for the object returned by the function, and *this* for the current class instance.

The *property* annotations are a way for the compiler to gather information about the member variables of the written-to container objects. These annotations are comma-separated lists of equations where the left-hand side of each equation is a member variable, and the right-hand sides are algebraic expressions of parameters or parameter member variables. The current system only handles equalities and not more complicated algebraic expressions, but this is not a difficult extension. These annotations are used for two purposes: constructing temporaries and fixing usages of the member variable. In order to handle both of these usages, each function is required to have *property* annotations for every property set in the constructor as well as every member variable used in any operator.

The *shape* annotation is an optional one, but it is essential for performing contraction and gaining the performance benefits associated with the elimination of compiler-generated temporaries. It consists of the *shape* keyword followed by a colon and a comma-separated list of container objects. It specifies which objects have the same shape as the result and therefore can be reused instead of generating a new compiler temporary. If the *shape* annotation is omitted, the compiler will assume that no container objects can be reused and will generate a new one for each operation.

The second type of annotation, the essential operation markers, mark the core operation of operator where the result is actually calculated and written. A key assumption of our system is that, since container classes encapsulate data arrays, all operators need to traverse their data and therefore

contain duplicate data traversal code. The interior of the data traversal loop is the main essential operation, and we rely on the programmer to distinguish between the unnecessary duplicated code and the essential operations. Essential operations are enclosed by *#pragma expfusion essentialop_begin X* and *#pragma expfusion essentialop_end X*, where X is an integer that is used to distinguish between different classes of essential operations. Multiple classes can come about in the case where multiple traversal loops exist or certain bits of code besides the interior of the traversal loop exist (e.g., an important sanity check on the properties of the operands). Every operator is required to have the same set of essential operation classes even if an operation is empty, and only one essential operation of each class should be defined in each operator.

6.2 Assumptions

To transform code correctly, we make a few assumptions about the container class:

- The class is opaque so that the data can only be read and modified through the class interface.
- Container objects and their data do not alias (e.g., submatrices). This ensures that the compiler can determine exactly when a specific object is being read or written.
- Data traversal is similar in each operator. More specifically, we require that loops are compatible and traverse the data in same direction.

- Two or more essential operations in the same function do not rely on each other's output.

Although these assumptions may be relatively strong, they are true for almost all of the container class implementations that we have encountered.

6.3 Analysis and Optimization

With the above information, the compiler is able to perform expression fusion and contraction automatically and without modification to the core source code. An outline of our solution is given below:

```
For each node in the dataflow graph,
  def = all operands indicated by the assign annotations
  use = all operands indicated by the access annotations
      + any matrix parameters, if this is not an
          annotated function
Perform dataflow analysis to find reaching definitions

Create a tree node for each expression
For each expression,
  For each access of the expression
    Add the access's associated tree node
        as a child of the current node
For each use of an expression
  If this use is an non-annotated function,
    add all used expression tree nodes
        to the set finalized_expressions
  If two or more definitions reach this use,
    add all definition tree nodes to the set finalized_expressions

For each node in finalized_expressions
  Create a new function with arguments equal to the union of all
    leaf nodes in the expression tree and any parameters in
```

```
  the tree that are not an access or assign
Perform modified Sethi-Ullman numbering
Copy in the essential operations
Fix properties
Recursively fuse function calls
```

6.3.1 Building the Expression Trees

The first step in the optimization is to perform a reaching definitions dataflow analysis for the container objects. When the code is translated to an intermediate representation in LLVM, long expressions are translated into their constituent operator functions. This intermediate representation makes single expressions indistinguishable from multiline expressions, which are not handled by ETs. To perform this analysis, we use *assign* annotations to initialize def sets and *access* annotations to initialize *use* sets. Uses of container objects that are not annotated functions are also included in the *use* sets as these are not operators and thus we can assume that they do not modify the internal data.

With information about reaching definitions, we can begin to build the expression trees. Each operation is a tree node with children representing each accessed operand to the operation, and these tree nodes are connected to form a forest. The roots of the forest are equal to operations that need to be computed, and the leaves are equal to either other roots or single matrices. Nodes that need to be calculated are called finalized expressions and become finalized if and only if they are used by unannotated functions, in which case the function cannot be further chained, or if they compete with another *def*

for a *use*. The latter case happens when a definition occurs within a branch and it is not possible to tell whether the fused expression should include the subtree rooted at the current definition or the subtree rooted at the alternate definition. After this algorithm completes, the set of finalized expressions contains the expressions that are used and need to be fused.

6.3.2 Fusing Essential Operations

For each finalized expression, we create a new function that will hold the fused expression. The fused function uses the function of the root of the expression tree as a base into which all the other essential operations are copied. The function's parameters change to include all leaf nodes in the entire expression tree as well as any parameters in the tree that are not part of an access or assign annotation.

For the simplest case, fusing essential operations simply involves post-order traversal of the expression tree for each essential operation class. The code between essential operation markers are copied in post-order into the fused function in order to ensure that values are written before they are used. To store the intermediate values in this computation, the naive approach is to create a temporary matrix for each edge in the expression tree and update references accordingly. This temporary matrix is created with the help of *property* annotations and the *temp_constructor* annotation. The *property* annotations can be recursively chained to generate an expression representing the temporary's properties as a function of actual parameters. These properties

are passed to the *temp_constructor* to generate the necessary temporary.

6.3.3 Eliminating Temporaries

The above description of essential operation fusion is able to fuse loops within these operations but not perform contraction to eliminate temporaries. The performance benefit of removing loop overheads is relatively small. In order to eliminate temporaries, we must be able to reuse existing temporaries, and in order for reuse to be legal, the shapes of the temporary container classes must be compatible. This is where the *shape* annotations become useful. The current operation's result can be stored in the same container object as any parameter listed in the *shape* annotation.

To order the operations and allocate temporary container objects in the most efficient manner, we note that this problem is very similar to the classic problem of register allocation for expressions. Register allocation for expressions is usually done through an algorithm called Sethi-Ullman numbering, which cannot be applied directly because it is designed with the assumption that all registers are equal. Our system calculates the most efficient usage of temporaries through a modified version of Sethi-Ullman numbering that generalizes the register allocation algorithm to allow for multiple incompatible types of storage. Each node maintains a list of tuples that represents the necessary temporaries for the expression rooted at that node.

1. Traverse the expression tree in postorder.

2. For every non-leaf node n that has no subexpressions, add (1, n) to the node's list of tuples.

3. For every other non-leaf node n:

   (a) For each node m_i referenced in the shape annotation, take its main tuple (x_i, m_i)

   (b) If no such m_i exists, add (1, n) to the node's list of tuples

   (c) Else let y be the maximum x_i of all such tuples. Add ($y + t - 1$, n) to the list, where t is the number of nodes with y as the first value in their main tuple

   (d) For all other tuples in each child, add them to the node's list of tuples

The above algorithm is able to assemble a list of types of temporaries along with the necessary number of each type, and there are several important differences from the original Sethi-Ullman numbering to pay attention to. First, instead of a single value, a list is maintained to keep track of all different types of temporaries, and the required temporaries for each type can only be incremented by a node with the same type. Second, leaves are not mangled in the calculation because they may be used in another expression. Third, we have generalized the original policy of incrementing by one if all operands have the same number to account for a variable number of operands. This algorithm increments by the number of nodes with the same, maximum value

minus one, which degenerates to the original policy in the case of an operation with exactly two operands of equal shape. Finally, reuse can only occur either between parent and child or transitively, but not, for example, across layers or across subtrees. This limitation is due to the fact that these cases are either purely coincidental or due to a complicated aspect of the operations that are not general enough to make a more complex annotation system profitable. The current system handles the common case, namely the arithmetic operations of arrays and matrices.

Additional policies for temporary allocation and operation ordering are possible as well. Future work includes extending other register allocation algorithms like a more advanced Sethi-Ullman numbering that takes advantage of operator associativity and commutativity or an algorithm that optimizes for parallelizability [8].

6.3.4 Function Calls and Properties

After the essential operations of an expression are fused into a single function, there may still be a few loose ends to tie up. A common tool for implementing operations is the use of function calls in the form of helper functions, recursive functions, or even get/set functions. Depending on the semantics of the function calls, calls from two operations can be fused in three different ways, and in general, the approach to take cannot be determined without programmer help. If the calls should be done sequentially, then programmers should treat them as regular essential operations, and they will be

copied into the fused function as such.

If the calls actually write to the result and therefore contain essential operations as well, then programmers should use a *#pragma expfusion essentialop_function X* annotation, where X is an essential operation class. (These are the same classes that are used by regular essential operations so that the compiler knows where to place subfunctions when fusing with functions that do not contain a call of that class). This annotation will tell the compiler to fuse the functions in the same manner that the parent function was fused. If the calls only read the result, then the functions do not actually need to be fused. If the function accesses the non-property data of the result before it is written, then this is a violation of our dependency assumptions and is therefore illegal. At most, the compiler needs to replace references to the result's properties with fused versions, which is described below.

Uses of properties of the function can occur for various reasons including loop/recursion termination conditions and sanity checks. The *property* annotations give the compiler an easy way to calculate any necessary property with only the properties of base operands to the expression. Any use of a non-leaf node's properties is recursively replaced with the programmer-supplied algebraic expression, providing a simple way to handle these uses.

6.4 Limitations

Our solution is able to perform expression fusion automatically and with minimal programmer annotations, but this smaller burden on the pro-

grammer comes with a few limitations. As ETs can perform arbitrary code generation and are Turing complete, our solution is clearly less powerful and less expressive. Our solution also requires a language extension, although it is backwards compatible with existing C++ code and programs that use our solution are backwards compatible with existing compilers as well.

The optimization mechanism of ETs is fundamentally different than that of our compiler solution, so they have a few benefits that we are not able to reproduce and vice versa. The expression fusion optimization in ETs is actually a utilization of a language feature called lazy evaluation. ETs force the programmer to specify the essential operation on a per element basis instead of the per operation basis that our solution requires. This means that, in addition to fusion of operations over whole matrices, ETs are able to fuse accesses to individual elements of the result without needing to calculate the rest of the result. Another benefit of ETs comes as a side effect of template specialization. Classes are duplicated for each possible permutation of template parameters, so each specialized class can use the template parameter as a constant. Specialization therefore leads to increased opportunities for constant propagation, which has been utilized for compiler-assisted loop unrolling and other compiler optimizations.

Chapter 7

Evaluation

We have implemented a compiler pass on top of the LLVM compiler infrastructure that can perform expression fusion using the method and annotations described above. It is a subset of the full implementation, but it is complete enough to correctly optimize simple matrix libraries. In this section we will focus on experimental comparison between clean, ET, and compiler-optimized programs that use similar matrix libraries. The current implementation of our solution requires translation from C++ to LLVM's intermediate representation, back to C, and then into executable format using gcc. These multiple steps can introduce inefficiencies into the code, and in order to be fair, the clean and ET versions have been translated through LLVM as well. Also, since ETs assume the presence of strong compiler support, our optimized version and the clean version have been compiled with the same optimizations. All tests have been performed on a machine with a 3.16GHz Intel(R) Core(TM)2 Duo CPU.

We have tested our optimization on a real problem, namely the Jacobi iteration method for solving systems of linear equations, and the results are graphed in Figure 7.1. Our solution has slightly better performance than that

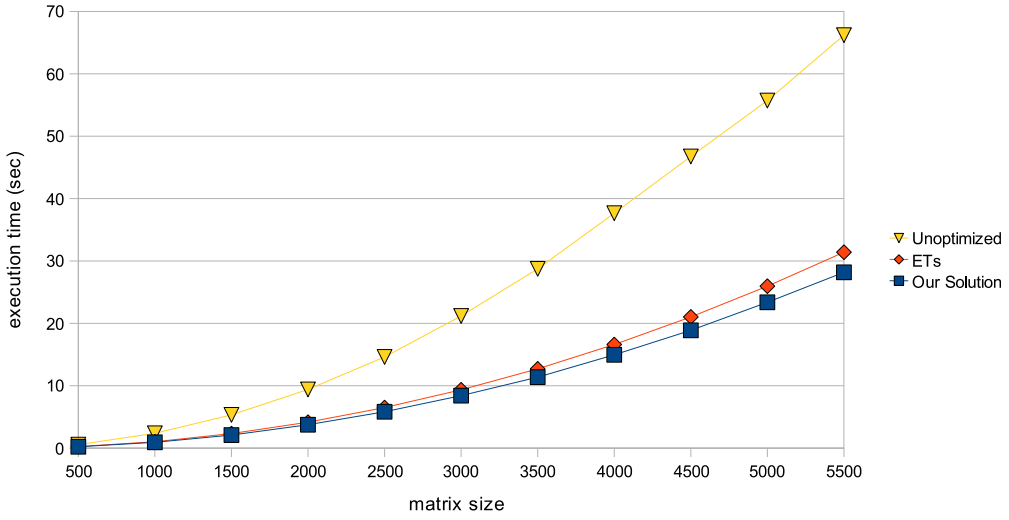
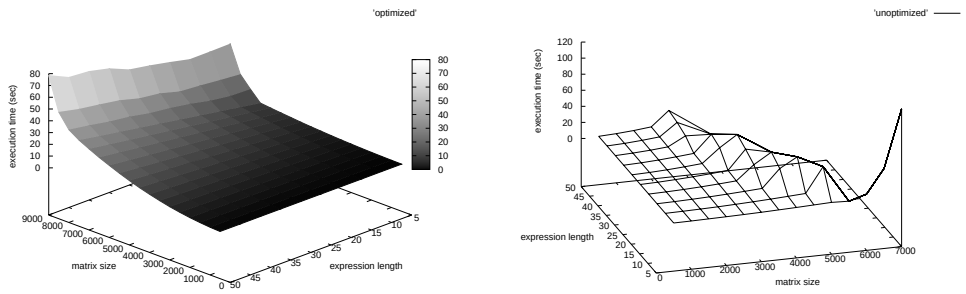


Figure 7.1: Execution time for all three approaches for Jacobi Iteration

of Expression Templates, which may be due to the code bloat present in ETs. These tests also show the more than 2× slowdown of regular C++ code that is not fused and contracted.

In addition to acheiving slightly better performance, our solution allows for a much simpler implementation of the library. The very simple vector library used in the original ET paper [15] has no other optimizations, but is still 302 lines of code. The equivalent library that uses our solution takes around 99 lines of code including annotations. This code bloat only gets worse as the library becomes more complicated. The overloaded operators of the Blitz++ library [16] that only generate the expression objects and do not implement the operations themselves take about 14000 lines of largely redundant code.



(a) our solution (b) unoptimized

Figure 7.2: Matrix size and expression length vs. execution time for (a) a clean library optimized by our solution and (b) an unoptimized version of the clean library

We have also tested our compiler optimization on a micro-benchmark consisting of matrix initializations and a single expression, and the results are graphed in Figure 7.2. The tests compare the number of operations in an expression and the size of the matrices versus execution time. For a program optimized with our compiler solution, the execution time increases mainly with matrix size and only slightly with expression size. The unoptimized program's execution time graph exhibits large jumps as we test larger matrix sizes/expression lengths, and it produces segmentation faults for the unplotted points. The reason why this happens is because the C++ standard states that temporary objects are destroyed as the last step in evaluating the full-expression that contains the point where they were created [1]. This means that, in the case of a 50-operand expression, all 50 temporaries will stay allocated until the computation is finished.

Chapter 8

Conclusion

Container classes are used for many domain-specific problems, which may be amenable to domain-specific optimizations. In the case of matrix libraries in C++, expression fusion and contraction is a necessary optimization for good performance, but most previous work in fusion have not accounted for the increasingly complex nature of container classes.

In this thesis, we have discussed the state of art in expression fusion, Expression Templates, and enumerated its drawbacks. We have also presented an alternative to ETs in the form of a compiler optimization that uses programmer annotations to fuse and contract complex operators. The optimization allows programs to design container classes in a straightforward manner, with the same cache locality benefits as ETs, but without most of the limitations.

Bibliography

[1] *ISO/IEC 14882:2003: Programming languages: C++.* American National Standards Institute, New York, NY, USA, 2nd edition, 2003.

[2] F. Bassetti, K. Davis, and D. Quinlan. C++ Expression Templates Performance Issues in Scientific Computing. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 635–639, 1998.

[3] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, pages 191–199, New York, NY, USA, 1993. ACM.

[4] Ganesh Bikshandi, Jia Guo, Christoph von Praun, Gabriel Tanase, Basilio B. Fraguela, María Jesús Garzarán, David A. Padua, and Lawrence Rauchwerger. Design and Use of htalib - A Library for Hierarchically Tiled Arrays. In *Proceedings of LCPC 2006 conference on Languages and Compilers for Parallel Computing*, pages 17–32, 2006.

[5] Shigeru Chiba. A Metaobject Protocol for C++. In *OOPSLA 1995 Conference Proceedings Object-Oriented Programming Systems, Languages, and Applications*, pages 285–299, October 1995.

[6] Guang R. Gao, R. Olsen, Vivek Sarkar, and Radhika Thekkath. Collective Loop Fusion for Array Contraction. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 281–295, London, UK, 1993. Springer-Verlag.

[7] Jochen Härdtlein, Alexander Linke, and Christoph Pflaum. Fast Expression Templates. In *International Conference on Computational Science (2)*, pages 1055–1063. Springer, 2005.

[8] Warren Hunt, Bertrand A. Maher, Doug Burger, and Kathryn S. Mckinley. Optimal Huffman Tree-Height Reduction for Instruction-Level Parallelism. Technical Report TR-08-34, Department of Computer Sciences, The University of Texas at Austin, 2009.

[9] Ken Kennedy and Kathryn S. McKinley. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 301–320. Springer Berlin / Heidelberg, 1994.

[10] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, University of Illinois at Urbana-Champaign, December 2002.

[11] E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The Implementation and Evaluation of Fusion and Contraction in Array Languages.

In *Proceedings of the ACM SIGPLAN '98 conference on programming language design and implementation*, pages 50–59, New York, NY, USA, June 1998. ACM.

[12] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. `C and tcc: a Language and Compiler for Dynamic Code Generation. *ACM Trans. Program. Lang. Syst.*, 21(2):324–369, 1999.

[13] William Pugh. Uniform Techniques for Loop Optimization. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 341–352, 1991.

[14] Sharad K. Singhai and Kathryn S. McKinley. A Parametrized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. *The Computer Journal*, 40(6):340–355, 1997.

[15] Todd L. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995.

[16] Todd L. Veldhuizen. Arrays in blitz++. In *In Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE98)*, pages 223–230, 1998.

[17] Todd L. Veldhuizen. C++ Templates are Turing Complete. Technical report, Indiana University, 2003.